Efficient Reasoning about Data Trees via Integer Linear Programming

Claire David, Leonid Libkin, Tony Tan

Efficient Reasoning about Data Trees via Integer Linear Programming

Claire David
Université Paris-Est
Claire.David@univ-mlv.fr

Leonid Libkin
University of Edinburgh
libkin@inf.ed.ac.uk

Tony Tan
University of Edinburgh
ttan@inf.ed.ac.uk

ABSTRACT

Data trees provide a standard abstraction of XML documents with data values: they are trees whose nodes, in addition to the usual labels, can carry labels from an infinite alphabet (data). Therefore, one is interested in decidable formalisms for reasoning about data trees. While some are known – such as the two-variable logic – they tend to be of very high complexity, and most decidability proofs are highly nontrivial. We are therefore interested in reasonable complexity formalisms as well as better techniques for proving decidability.

Here we show that many decidable formalisms for data trees are subsumed – fully or partially – by the power of tree automata together with set constraints and linear constraints on cardinalities of various sets of data values. All these constraints can be translated into instances of integer linear programming, giving us an NP bound on the complexity of the reasoning tasks. We prove that this bound, as well as the key encoding technique, remain very robust, and allow the addition of features such as counting of paths and patterns, and even a concise encoding of constraints, without increasing the complexity. We also relate our results to several reasoning tasks over XML documents, such as satisfiability of schemas and data dependencies and satisfiability of the two-variable logic.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; G.1.6 [**Numerical Analysis**]: Optimization—*Integer programming*; H.2.1 [**Database Management**]: Logical Design—*Data Models*

General Terms

Algorithms, Theory

ICDT 2011 March 21-23, 2011, Uppsala, Sweden.

Keywords

XML, tree languages, data values, Presburger arithmetic, reasoning, integer linear programming

1. INTRODUCTION

Traditional approaches to studying logics on trees use a finite alphabet for labeling tree nodes. The interest in such logics was reawakened by the development of XML as the standard for data exchange on the Web. Logical formalisms provide the basis for query languages as well as for reasoning tasks, including many static analysis questions such as consistency of specifications, query optimization, and typing [1, 2, 13, 14, 22, 29].

The simplest abstraction of XML documents is ordered unranked finite trees whose nodes are labeled by letters from a finite alphabet [23, 34]. This abstraction works well for reasoning about structural properties, but real XML documents carry *data*, which cannot be captured by a finite alphabet. Thus, there has been a consistent interest in *data trees*, i.e., trees in which nodes carry both a label from a finite alphabet and a data value from an infinite domain [6, 7, 8, 12, 25, 17]. It seems natural to add at least the equality of data values to a logic over data trees. But while for finitely-labeled trees many logical formalisms are decidable by converting formulae to automata (e.g., the monadic second-order logic MSO), adding data-equality makes even FO (first-order logic) undecidable.

This explains why the search for decidable reasoning formalisms over data trees has been a common theme in XML research. Such a search has largely followed two routes. The first takes a specific XML reasoning task, or a set of similar tasks, and builds algorithms for them (see, e.g, [2, 3, 5, 9, 13, 29, 14]). The second attempts to find a sufficiently general logical formalism that is decidable, and can express some XML reasoning tasks of interest (see, e.g, [12, 6, 16]).

While both approaches have yielded many nontrivial and influential results, they are not completely satisfactory. The first approach gives us specialized algorithms for concrete problems, but no general tools. The second approach tends to produce extremely high complexity bounds, such as 4EXPTIME, or even non-primitive-recursive, and in addition, the

proofs are usually highly nontrivial and are very hard to adapt to other reasoning tasks.

Instead we want a sufficiently general formalism – in fact, a family of formalisms, that are not extremely complicated to deal with, and at the same time give us acceptable complexity bounds. For reasoning tasks (as opposed to, say, query evaluation which we are used to in databases), acceptable complexity is often viewed as single-exponential [28], or better yet, NP (since SAT solvers are now a practical tool for many static analysis problems [20]).

The particular class of formalisms we deal with here is motivated by several concrete reasoning tasks studied in the XML context, as well as some decidable logical formalisms. We now briefly describe those. One of the earliest reasoning problems studied in the XML context was the problem of reasoning about *keys and inclusion constraints*. An XML key says that for a given label a, the data value of an a-node (i.e., node labeled a) uniquely determines the node. An inclusion constraint says that every data value of an a-node will occur in a b-node as well. Such constraints are typical in databases, from which many XML documents are generated. The question is then whether they are consistent with the schema of an XML document, usually given as a tree automaton, or a DTD. This problem is decidable in NP [13].

On the logic side, there appear to be two main ideas leading to decidability. One starts with a temporal logic, and adds a limited memory for keeping and comparing data values. Examples include [12, 16], but the logics, although decidable, have extremely high complexity (non-primitive-recursive). A different approach based on restricting the number of variables was followed by [6], which showed that FO^2, first-order logic with two variables, is decidable over data trees. In fact, even $\exists\mathrm{MSO}^2$, its extension with existential monadic second-order quantifiers, is decidable. The complexity drops to elementary but is still extremely high: the decision procedure runs in 3NEXPTIME.

Our formalisms extend the specific constraints such as keys and inclusions, and yet come very close to subsuming the power of logics such as $\exists\mathrm{MSO}^2$, while permitting many properties which are not even definable in MSO. To motivate the kind of constraints we use, let us restate keys and inclusion constraints in a slightly different way. For this, we need two new notations: $V(a)$ stands for the set of data values in a-labeled nodes, and $\#a$ is the number of a-nodes.

- A key simply states that $\#a = |V(a)|$. We view this as a *linear constraint*, and allow arbitrary linear constraints over the values $\#a$ and $|V(a)|$, for example, $|V(a)| \geq 2|V(b)| - \#c$.
- An inclusion constraint states that $V(a) \subseteq V(b)$, or, equivalently, $V(a) \cap \overline{V(b)} = \emptyset$. We generalize this to arbitrary *set-constraints* [26], stating that a Boolean combination of $V(a)$'s is either empty or nonempty.

We consider the problem of satisfiability of such constraints with respect to a schema declaration, given by an unranked tree automaton [21]. Or, formally: Given an unranked tree automaton $\mathcal{A}$ and a collection $\mathcal{C}$ of set and linear constraints, does there exist a tree t accepted by $\mathcal{A}$ that satisfies all the constraints in $\mathcal{C}$?

We prove that this problem, and several of its variations, are all decidable in NP (the hardness result has been known even for simple keys and inclusion constraints [13]). The techniques are all based on reduction to instances of *integer linear programming*. In fact, the basic result, unlike many decidability proofs [6, 12, 13], is quite easy to establish.

Already our basic result subsumes, not only reasoning about integrity constraints in XML (as in, e.g., [2, 13]), but also a very large fragment of $\exists\mathrm{MSO}^2$. These relationships will be made precise in Section 6. Note that even the decidability of the satisfiability problem does *not* follow from known results such as [6] which are restricted to fragments of MSO; in contrast, our formalism expresses many properties not definable in MSO.

One benefit of having simple proof techniques is that the basic result can be extended in several ways. One of the extensions introduces variables that count not just the number of nodes labeled a, but also the number of such nodes that permit some paths starting from them. For example, we can reason about the number of a-nodes that have a b-parent and a c-sibling. By extending the translation into integer linear programming, we obtain such extensions quite easily.

A more surprising extension is to *concisely represented* constraints. One way to reduce the size of the representation of linear constraints is to discard all zero entries from matrices defining them. This can shrink the size of the instance of the problem exponentially. A common phenomenon in complexity theory is that such a shrinking increases the complexity by an exponent. So more naturally, the expected bound thus would be NEXPTIME. We prove that, quite surprisingly, under such concise representation of constraints, the problem stays in NP.

As a final contribution, we make the entire approach completely algorithmic, by providing simple and self-contained translations: (1) from an unranked tree automaton $\mathcal{A}$ to a simple instance $\Psi_{\mathcal{A}}$ of linear programming, i.e. instances with only 0, 1 or -1 coefficients; and (2) from a solution of $\Psi_{\mathcal{A}}$ to a tree accepted by $\mathcal{A}$. Of course the existence of translation (1) is known [33, 18]; the contribution here is to provide a simple algorithm for doing it, that also results in simple formulae which could be used and implemented for concrete verification problems.

Organization. Section 2 presents the main definitions. In Section 3 we define the contraints and the problem we are interested in. In Section 4 we establish the basic result. An extension is presented in Section 5. In Section 6 we relate set and linear constraints to XML reasoning tasks and the logic $\exists\mathrm{MSO}^2$ of [6]. We study the complexity of the problem in the case of concise representation of the constraints in Section 7. Finally, the translation from unranked tree automata to linear integer programming is provided in Section

8. Some proofs are only sketched, or completely omitted, they will be available in the journal version.

2. PRELIMINARIES

Trees and automata. We start with the standard definitions of unranked finite trees and logics and automata for them. An unranked finite tree domain is a prefix-closed finite subset D of $\mathbb{N}^*$ (words over $\mathbb{N}$) such that $v \cdot i \in D$ implies $v \cdot j \in D$ for all $j < i$ and $v \in \mathbb{N}^*$. Given a finite labeling alphabet Σ, a Σ-labeled unranked tree is a structure $\langle D, E_{\downarrow}, E_{\rightarrow}, \{a(\cdot)\}_{a\in\Sigma}\rangle$, where

- D is an unranked tree domain,
- $E_{\downarrow}$ is the child relation: $(v, v \cdot i) \in E_{\downarrow}$ for $v \cdot i \in D$,
- $E_{\rightarrow}$ is the next-sibling relation: $(v \cdot i, v \cdot (i+1)) \in E_{\rightarrow}$ for $v \cdot (i+1) \in D$, and
- the $a(\cdot)$'s are labeling predicates, i.e. for each node v, exactly one of $a(v)$, with $a \in \Sigma$, is true.

The label of the node v in t will be denoted by $\ell ab_t(v)$, and the domain D is denoted by $\mathsf{Dom}(t)$.

Let r be a designated symbol in Σ. We assume that the root of the tree (i.e., the empty word) is labeled r, and no other node is labeled r. (This is not a restriction as we can always put a new root with a given label.)

An *unranked tree automaton* [10, 32] over Σ-labeled trees is a tuple $\mathcal{A} = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^{(Q^*)}$ is a transition function; we require that $\delta(q, a)$'s be regular languages over Q for all $q \in Q$ and $a \in \Sigma$.

A run of $\mathcal{A}$ over a tree t is a function $\rho_{\mathcal{A}} : \mathsf{Dom}(t) \to Q$ such that for each node v with n children $v \cdot 0, \ldots, v \cdot (n-1)$, the word $\rho_{\mathcal{A}}(v \cdot 0) \cdots \rho_{\mathcal{A}}(v \cdot (n-1))$ is in the language $\delta(\rho_{\mathcal{A}}(v), \ell ab_t(v))$. Of course, for a leaf v labeled a this means that v could be assigned state q iff the empty word ϵ is in $\delta(q, a)$. A run is accepting if $\rho_{\mathcal{A}}(\epsilon) \in F$, i.e., if the root is assigned an accepting state. A tree t is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The set of all trees accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$.

Data trees. In a data tree, besides carrying a label from the finite alphabet Σ each non-root node also carries a data value from some countably infinite data domain (to be concrete, we assume it to be $\mathbb{N}$). For a node v of a data tree t, labeled with a symbol $a \in \Sigma - \{r\}$, the assigned data value is denoted by $\mathrm{val}_t(v)$. We also denote the set of all data values assigned to a-nodes by $V_t(a)$. That is, $V_t(a) = \{\mathrm{val}_t(v) \mid \ell ab_t(v) = a\}$.

Note that we assume $V_t(r) = \emptyset$ (i.e., no data value assigned to the root). This is done for convenience and is not a restriction, as one can always add a new root without a data value.

Integer linear programming and Presburger formulae. Recall that an instance of integer linear programming consists of an $m \times k$ integer matrix $\mathbf{A}$ and a vector $\mathbf{b} \in \mathbb{Z}^m$. The question is whether there is a k-vector $\bar{v}$ over integers such that $\mathbf{A}\bar{v} \geq \mathbf{b}$.

The problem is well-known to be NP-complete. It is NP-hard even when entries are restricted to be 0s and 1s. Membership in NP follows from the fact that if $\mathbf{A}\bar{v} \geq \mathbf{b}$ has an integer solution, then there is one in which all entries are bounded by $(ak)^{p(m)}$, where a is the maximum absolute value that occurs in $\mathbf{A}$ and $\mathbf{b}$, and p is a polynomial [27].

We also consider existential Presburger formulae, i.e., first-order formulae over the structure $\langle \mathbb{Z}, +, 0, 1, < \rangle$. Such formulae can always be converted to formulae of the form

$$\varphi(\bar{x}) \;=\; \exists \bar{y}\ \mathrm{PBC}(\mathbf{A}_i \bar{v}_i \geq \mathbf{b}_i), \tag{1}$$

where PBC means a positive Boolean combination, and each $\mathbf{A}_i \bar{v}_i \geq \mathbf{b}_i$ is an instance of integer linear programming with variables $\bar{v}_i$ coming from $\bar{x}, \bar{y}$. Whenever we refer to existential Presburger formulae, we assume that they are in the form (1). We also only work with non-negative integers for $\bar{x}$ and $\bar{y}$, so we always assume that all the conditions $x_j \geq 0, y_l \geq 0$ are included in formulae. However, it is to be noted that the entries in $\mathbf{A}_i$ and $\mathbf{b}_i$ can be negative.

Notice that we occasionally use conditions such as $x > 0$ or $x + y \leq b$, or $x = y$, but these are easily put in the form (1) by changing them to $x \geq 1$, or $-x - y \geq -b$, or the conjunction of $x \geq y$ and $y \geq x$, respectively.

Satisfiability of existential Presburger formulae is known to be in NP. This follows from the bound of [27]. If we have a witness $(\bar{x}, \bar{y})$ for φ of the form (1), then it does so by making some of the $\mathbf{A}_i \bar{v}_i \geq \mathbf{b}_i$ true. Using the bound of [27], we see that no matter which combination of these instances of linear integer programming makes the formula true, there is a witness that will require only polynomially many bits in terms of the size of the formula. This implies the NP bound.

3. CONSTRAINTS AND THE SATISFIABILITY PROBLEM

In this section we give the precise definitions of set and linear constraints, and state the main satisfiability problem.

Set constraints. Recall that Σ is the labeling alphabet with a special symbol r for the root. *Data-terms* (or just terms) are given by the grammar

$$\tau := V(a) \mid \tau \cup \tau \mid \tau \cap \tau \mid \overline{\tau} \qquad \text{for } a \in \Sigma - \{r\}.$$

The semantics $[\![\tau]\!]_t$ is defined with respect to a data tree t:

- $[\![V(a)]\!]_t = V_t(a)$;
- $[\![\tau_1 \cap \tau_2]\!]_t = [\![\tau_1]\!]_t \cap [\![\tau_2]\!]_t$;
- $[\![\tau_1 \cup \tau_2]\!]_t = [\![\tau_1]\!]_t \cup [\![\tau_2]\!]_t$;
- $[\![\overline{\tau}]\!]_t = V_t - [\![\tau]\!]_t$;

where $V_t = \bigcup_{a\in\Sigma-\{r\}} V_t(a)$ is the set of data values found in the data tree t.

A *set constraint* is either $\tau = \emptyset$ or $\tau \neq \emptyset$, where τ is a term. A tree t satisfies $\tau = \emptyset$, written as $t \models \tau = \emptyset$, iff $[\![\tau]\!]_t = \emptyset$ (and likewise for $\tau \neq \emptyset$).

Note that set constraints $\tau_1 \subseteq \tau_2$ and $\tau_1 \subset \tau_2$ can be similarly defined, but they are easily expressible with the emptiness constraints (e.g., $\tau_1 \subseteq \tau_2$ means that $\tau_1 \cap \overline{\tau_2}$ is empty).

In particular, the inclusion constraint from the introduction is an example of a set constraint: to say that all data values of a-nodes occur as data values of b-nodes, we write $V(a) \cap \overline{V(b)} = \emptyset$.

Linear data constraints. Fix variables x_a for each $a \in \Sigma$ and z_S for each $S \subseteq \Sigma - \{r\}$. Linear data constraints are linear constraints over these variables.

The interpretation of x_a is $\#a(t)$ – the number of a-nodes in t. The interpretation of z_S is the cardinality of the set

$$[S]_t = \bigcap_{a \in S} V_t(a) \cap \bigcap_{b \notin S} \overline{V_t(b)}.$$

That is, $[S]_t$ contains data values which are found among a-nodes for all $a \in S$ but which are not attached to any b-nodes for the label $b \notin S$. Note that the sets $[S]_t$'s are disjoint, and that

$$V_t(a) = \bigcup_{S \text{ such that } a \in S} [S]_t.$$

This gives us much more information that just the number of data values in a-nodes, $|V_t(a)|$, which is simply

$$|V_t(a)| = \sum_{S \text{ such that } a \in S} z_S.$$

For instance, with such constraints we can reason about data values that occur in a- and c-nodes but do not occur in b-nodes: the number of those is simply $\sum\{z_S \mid a, c \in S,\ b \notin S\}$.

Notice that key constraints from the introduction are examples of linear data constraints; that the data values of a-nodes form a key is that the number of a-nodes, which is x_a, is equal to the number of data values found in the a-nodes, which is $|V_t(a)|$. It is expressible by the linear constraint:

$$x_a = |V_t(a)| = \sum_{S \text{ such that } a \in S} z_S.$$

We shall view linear data constraints as an instance of integer linear programming. If $\Sigma = \{a_1, \ldots, a_\ell\}$ and $S_1, \ldots, S_k$ is an enumeration of nonempty subsets of $\Sigma - \{r\}$ (thus $k = 2^{|\Sigma|-1} - 1$), then a set of m linear data constraints is $\mathbf{A}\bar{v} \geq \mathbf{b}$, where $\mathbf{A}$ is an $m \times (\ell + k)$-matrix over $\mathbb{Z}$ and $\mathbf{b} \in \mathbb{Z}^m$. It is satisfied in a data tree t if it is true when v is interpreted as the vector

$$\big(\#a_1(t), \ldots, \#a_\ell(t),\ \big|[S_1]_t\big|,\ \ldots,\ \big|[S_k]_t\big|\big).$$

Satisfiability problem. Let $\mathcal{C}$ denote a collection of set and linear data constraints. If a tree t satisfies all the constraints in $\mathcal{C}$, we write $t \models \mathcal{C}$. We study the following satisfiability problem.

PROBLEM:	SAT($\mathcal{A}, \mathcal{C}$)
INPUT:	an unranked tree automaton $\mathcal{A}$; a collection $\mathcal{C}$ of set and linear data constraints
QUESTION:	is there a tree t accepted by $\mathcal{A}$ such that $t \models \mathcal{C}$?

The problem of consistency of XML constraints and schemas [2, 13] is a special instance of this problem. We shall later see that other problems related to XML constraints, as well as a large fragment of the two-variable logic can be formulated as special cases of SAT($\mathcal{A}, \mathcal{C}$). Moreover, SAT($\mathcal{A}, \mathcal{C}$) includes many instances that cannot even be formulated in MSO, which is the logic that typically subsumes XML reasoning tasks (for example, the linear constraint which states that $\#a(t) > 2 \cdot \#b(t)$ is not expressible in MSO, but is a simple linear data constraint $x_a > 2x_b$).

4. DECIDING SATISFIABILITY

We shall now prove the decidability of SAT($\mathcal{A}, \mathcal{C}$) problem. In our first result, we assume a simple way of measuring the size of the input:

- For the automaton $\mathcal{A}$, we take the size of the transition table, where each transition $\delta(q, a)$ is represented by an NFA (or by a regular expression, since an NFA can be computed from it in polynomial time).
- The size of each set constraint $\tau \{=, \neq\} \emptyset$ is measured as the size of the parse-tree for the term τ.
- The size of the linear data constraints $\mathbf{A}\bar{v} \geq \mathbf{b}$ is the sum of sizes of $\mathbf{A}$ and $\mathbf{b}$, with numbers represented in binary.

THEOREM 4.1. *The problem* SAT($\mathcal{A}, \mathcal{C}$) *is solvable in* NP.

Before proving this result, we give a couple of remarks. First, hardness for NP has been known, as it easily follows from the hardness result for XML keys and foreign keys in [13] and many other proofs can be adapted as well. In this result the most important task is to prove the upper bound, showing that reasoning tasks have acceptable complexity.

Second, extending the class of linear constraints by adding multiplication leads to the immediate loss of decidability, as Hilbert's 10th problem can be trivially encoded. This remains undecidable even for quadratic equations. On the other hand, if we extend the class of linear constraints to *Prequadratic Diophantine Equation*, where in addition to linear constraint, constraints such as $x_i \leq x_j x_k$ are allowed, it becomes decidable in NEXPTIME [15].

Proof of Theorem 4.1. Let $\Sigma = \{a_1, \ldots, a_n\}$ and $S_1, \ldots, S_k$ be the enumeration of non-empty subsets of $\Sigma - \{r\}$. We fix the vectors $\bar{x} = (x_{a_1}, \ldots, x_{a_n})$ and $\bar{z} = (z_{S_1}, \ldots, z_{S_k})$.

We first show how to express set constraints in terms of linear data constraints. For a term τ, we define a family $\mathbb{S}(\tau)$ of subsets of Σ as follows.

- If $\tau = V(a)$, then $\mathbb{S}(\tau) = \{S \mid a \in S \text{ and } S \subseteq \Sigma - \{r\}\}$.
- If $\tau = \overline{\tau}_1$, then $\mathbb{S}(\tau) = 2^{\Sigma - \{r\}} - \mathbb{S}(\tau_1)$.
- If $\tau = \tau_1 \star \tau_2$, then $\mathbb{S}(\tau) = \mathbb{S}(\tau_1) \star \mathbb{S}(\tau_2)$, where $\star$ is $\cap$ or $\cup$.

It follows immediately that for every data tree t, we have $[\![\tau]\!]_t = \bigcup_{S \in \mathbb{S}(\tau)} [S]_t$. Recall that the sets $[S]_t$'s are disjoint. Hence, the set constraint $\tau = \emptyset$ can be expressed as a linear data constraint $\sum_{S \in \mathbb{S}(\tau)} z_S = 0$. Similarly, $\tau \neq \emptyset$ can be expressed as $\sum_{S \in \mathbb{S}(\tau)} z_S \geq 1$. Since the size of linear constraints is exponential in Σ, this transformation is polynomial in the size of the whole input.[1] Hence, from now on, we can assume that the set $\mathcal{C}$ is of the form $\mathbf{A}(\bar{x}, \bar{z}) \geq \mathbf{b}$, and thus is given by a quantifier-free Presburger formula $\psi_{\mathcal{C}}(\bar{x}, \bar{z})$.

Next, we convert automata into linear constraints. In [33] it is shown that given a context free grammar G, whose terminals are $a_1, \ldots, a_n$, one can construct in polynomial time an existential Presburger formula $\varphi_G(x_1, \ldots, x_n)$ such that $\varphi_G(m_1, \ldots, m_n)$ holds if and only if there exists a word $w \in \mathcal{L}(G)$ such that $\#a_1(w) = m_1, \ldots, \#a_n(w) = m_n$, where $\#a_i(w)$ denotes the number of occurrences of a_i in the word w. Then, in [18] it is observed that the method can be extended to ranked tree automata. Since unranked tree automata can be easily converted to ranked tree automata with additional new symbol, we can construct the existential Presburger formula $\varphi_{\mathcal{A}}(x_1, \ldots, x_n)$ for unranked tree automaton $\mathcal{A}$, with one extra existential quantifier for the new symbol[2]. Hence, we have:

LEMMA 4.2. (See also Section 8.) *Given an unranked tree automaton $\mathcal{A}$, over alphabet $\Sigma = \{a_1, \ldots, a_n\}$, one can construct in polynomial time an existential Presburger formula $\varphi_{\mathcal{A}}(x_1, \ldots, x_n)$ such that if $w \in \mathcal{L}(\mathcal{A})$, then $\varphi_{\mathcal{A}}(\#a_1(w), \ldots, \#a_n(w)$ holds; and conversely, if $\varphi_{\mathcal{A}}(m_1, \ldots, m_n)$ holds, then there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $\#a_1(t) = m_1, \ldots, \#a_n(t) = m_n$.*

Going back to the proof of Theorem 4.1, we introduce additional variables v_a for each $a \in \Sigma - \{r\}$. The intended meaning of v_a is the cardinality of $V_t(a)$. Let $\bar{v}$ be the vector $(v_{a_1}, \ldots, v_{a_n})$. We next define two formulae that ensure proper interaction between $\psi_{\mathcal{C}}$ and $\varphi_{\mathcal{A}}$. First,

$$\chi(\bar{v}, \bar{x}, \bar{z}) = \bigwedge_{a \in \Sigma - \{r\}} (v_a = \sum_{a \in S} z_S) \wedge (v_a \leq x_a)$$

states the expected conditions on these variables, given their intended interpretations. Second,

$$\chi'(\bar{v}, \bar{x}) = \bigwedge_{a \in \Sigma - \{r\}} (x_a = 0 \vee v_a > 0)$$

ensures that if a-nodes exist (i.e., $x_a > 0$), then at least one data value is found in the a-nodes.

We now consider a Presburger formula $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$

$$\exists \bar{v} \left(\psi_{\mathcal{C}}(\bar{x}, \bar{z}) \wedge \varphi_{\mathcal{A}}(\bar{x}) \wedge \chi(\bar{v}, \bar{x}, \bar{z}) \wedge \chi'(\bar{x}, \bar{z})\right).$$

To convert $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$ into the form (1), we simply move all the existential quantifier in $\varphi_{\mathcal{A}}(\bar{x})$ to the front. Each atomic predicate inside $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$ can then be viewed as an instance of integer linear programming $\mathbf{A}\bar{y}_i \geq \mathbf{b}_i$.

LEMMA 4.3. *Given tuples of non-negative integers $\bar{n} = (n_a)_{a \in \Sigma}$ and $\bar{m} = (m_S)_{S \subseteq \Sigma - \{r\}}$, the formula $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{n}, \bar{m})$ holds iff there exists a data tree t accepted by $\mathcal{A}$ such that*

1. *$n_a = \#a(t)$ for each $a \in \Sigma - \{r\}$;*
2. *$m_S = |[S]_t|$ for each $S \subseteq \Sigma - \{r\}$;*
3. *$t \models \mathcal{C}$.*

PROOF. The "if" part is immediate from the construction of $\Psi_{(\mathcal{A},\mathcal{C})}$. We prove the "only if" direction. Suppose $\Psi(\bar{n}, \bar{m})$ holds. That is, there exists a witness $\bar{v}$ such that

$$\varphi_{\mathcal{C}}(\bar{n}, \bar{m}) \wedge \varphi_{\mathcal{A}}(\bar{n}) \wedge \chi(\bar{v}, \bar{n}, \bar{m}) \wedge \chi'(\bar{n}, \bar{m}) \text{ holds.}$$

Since $\varphi_{\mathcal{A}}$ holds, by Lemma 4.2, there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $(\#a_1(t), \ldots, \#a_n(t)) = \bar{n}$.

Now we show how to assign data values to the nodes in the tree t so that in the resulting data tree t' we have $m_S = |[S]_{t'}|$, for every $S \subseteq \Sigma - \{r\}$. Let $K = \sum_{S \subseteq \Sigma - \{r\}} m_S$, and we shall use the set $\{1, \ldots, K\}$ as the data values. Let

$$\xi : \{1, \ldots, K\} \mapsto 2^{\Sigma - \{r\}} - \emptyset$$

be a function satisfying $|\xi^{-1}(S)| = m_S$, for each $S \subseteq \Sigma - \{r\}$. The witness for $\bar{v}$ is $(\sum_{a_2 \in S} m_S, \ldots, \sum_{a_n \in S} m_S)$.

The data tree t' is obtained by letting $V_{t'}(a)$ be $\bigcup_{a \in S} \xi^{-1}(S)$. This is possible since $\chi(\bar{v}, \bar{n}, \bar{m})$ holds as $\sum_{a \in S} |\xi^{-1}(S)| = v_a \leq \#a(t) = n_a$. By definition of the function ξ, we obtain that $[S]_{t'} = \xi^{-1}(S)$, thus, $|[S]_{t'}| = m_s$, for each $S \subseteq \Sigma - \{r\}$. This proves the lemma. □

We now have an NP algorithm for SAT$(\mathcal{A}, \mathcal{C})$: in polynomial time we construct the formula $\Psi_{(\mathcal{A},\mathcal{C})}(\bar{x}, \bar{z})$ and then check for its satisfiability. It runs in NP, and Lemma 4.3 implies that it solves SAT$(\mathcal{A}, \mathcal{C})$. □

We shall see in the next section that our algorithm for SAT$(\mathcal{A}, \mathcal{C})$ gives some results obtained by using much harder techniques (such as reasoning about constraints in XML), and comes very close to giving us results obtained by *considerably much harder* techniques (like the results of [6]). The simpler structure of the proof will lead to some extensions that otherwise would have been very hard to obtain.

[1] In Section 7 when we look at the concise representations of the input, we will need a more refined technique for eliminating set constraints.

[2] We shall present a more thorough construction in Section 8.

5. AN EXTENSION: COMPLEX PROPERTIES OF NODES

We now demonstrate how the simple structure of the proof lets us obtain extensions for the main reasoning task almost effortlessly.

So far we were counting numbers of nodes $\#a(t)$ – i.e., nodes labeled a. Checking whether a node is labeled a is a simple property expressed by a fixed MSO (in fact, by an atomic FO) formula with one free variable. We now show that we can count sets of nodes expressed by arbitrary fixed MSO formulae and use them in linear constraints.

More precisely, let $\pi(x)$ be an MSO formula with one free first-order variable in the usual vocabulary of unranked trees, that is, $E_{\downarrow}, E_{\rightarrow}$, and $a(\cdot)_{a\in\Sigma}$ for child and next-sibling edges and labeling predicates. These formulae select nodes in trees. We let $\#\pi(t)$ be the cardinality of the set of nodes in t that satisfy π.

Using our proof, we can extend the decidability result to constraints that include "counting" the number of nodes output selected by such formulae $\pi(x)$. Note that unary MSO subsumes many XML formalisms, for example node formulae of XPath (or even conditional XPath).

If $\Pi = \{\pi_1(x), \ldots, \pi_s(x)\}$ is collection of such MSO formulae, then we refer to Π-linear constraints: these are linear constraints over the usual variables x_a's and z_S's, as well as w_{π_i}'s, interpreted as $\#\pi_i(t)$. We then deal with the problem Π-SAT$(\mathcal{A},\mathcal{C})$: its input is an automaton $\mathcal{A}$ and a collection $\mathcal{C}$ of set and Π-linear constraints, and the question is whether these are satisfiable.

Our proof immediately implies that the problem is decidable:

COROLLARY 5.1. *The problem* Π-SAT$(\mathcal{A},\mathcal{C})$ *is decidable.*

PROOF. We can embed the formulae $\pi_1, \ldots, \pi_s$ into the automaton $\mathcal{A}$ and check the existence of a tree over the alphabet $\Sigma \times 2^{\Pi}$, where *(i)* its Σ projection is accepted by $\mathcal{A}$ and *(ii)* for each node labeled with $(a, P) \in \Sigma \times 2^{\Pi}$, a formula π is satisfied iff $\pi \in P$ is satisfied. The linear constraints in $\mathcal{C}$ over the variables x_a's and z_S's can be easily converted into the variables $x_{a,P}$'s and z_T, where $P \subseteq 2^{\Pi}$ and $T \subseteq (\Sigma \times 2^{\Pi})$. □

The complexity of Π-SAT$(\mathcal{A},\mathcal{C})$ of course depends on how the formulae $\pi_1, \ldots, \pi_n$ are given. If they are given as MSO formulae, then it is immediately known that the complexity is non-elementary. But these formulae are also captured by the *query automata* of [24]: these are automata that also select nodes in their accepting runs. With query automata, the complexity drops to NEXPTIME, and in some cases to NP.

COROLLARY 5.2. *1. If the formulae in* Π *are given as query automata, then* Π-SAT$(\mathcal{A},\mathcal{C})$ *is decidable in* NEXPTIME.

2. Moreover, it is decidable in NP *if* Π *is fixed, or even if for each symbol* $a \in \Sigma$ *the number of formulae* $\pi_i(x)$ *which can be true in* a*-nodes is fixed.*

PROOF. Item (1) is straightforward, as the non-elementary blow-up for SAT$(\mathcal{A},\mathcal{C})$ occurs in translating the MSO formulae to query automata. However, the blow-up from NP (complexity of SAT$(\mathcal{A},\mathcal{C})$) to NEXPTIME occurs when moving from the alphabet Σ to $\Sigma \times 2^{\Pi}$. Thus, if Π is fixed, then the complexity remains in NP.

Moreover, if for each symbol $a \in \Sigma$ the number of formulae $\pi_i(x)$ which can be true in a-nodes is fixed, we do not need to move to the alphabet $\Sigma \times 2^{\Pi}$. We can stay in the alphabet Σ, and embed each $\pi \in \Pi$ inside the automaton $\mathcal{A}$. The automaton $\mathcal{A}$ can remember the fixed number of nodes that satisfy π and verify that indeed such is the case. This way we avoid the exponential blow-up and remains in NP. □

While converting from MSO to query automata is non-elementary, for some other formalisms that complexity is much lower: for example, [19] shows how to convert conditional-XPath to query automata in single-exponential time.

6. COMPARISON WITH OTHER FORMALISMS

We now show how the satisfiability problem SAT$(\mathcal{A},\mathcal{C})$ relates to reasoning tasks for XML with data.

6.1 XML constraints

As we already noticed, keys and inclusion constraints, studied extensively in the XML context (and included in the standards) are modeled with set and linear constraints. A simple key, saying that data values determine a-nodes, is a linear constraint $x_a = \sum_{a\in S} z_S$, and an inclusion constraint saying that data values of a-nodes occur as data values of b-nodes is $V(a) \cap \overline{V(b)} = \emptyset$. Similarly, one can handle *denial constraints*, often used in dealing with inconsistent data. An example of a denial constraint is saying that the same data value cannot appear in both an a-node and a b-node; this is expressible as $V(a) \cap V(b) \neq \emptyset$.

Our result implies that the satisfiability problem for key, inclusion, and denial constraints wrt an automaton is solvable in NP. Note however that to express a key as a linear constraint one needs exponentially many (in Σ) variables z_S, while we can compactly encode keys simply by letters involved in them, requiring $\log|\Sigma|$ bits instead. It turns out that this does not change the bound for keys and inclusion constraints; our proof can easily be adjusted to show:

COROLLARY 6.1. *The satisfiability problem for key (encoded by* $\log|\Sigma|$ *bits) and inclusion constraints wrt an automaton is solvable in* NP.

PROOF. Let $\mathcal{A}$ be an automaton over the alphabet Σ and let $\mathcal{C}$ be a collection of keys and inclusion constraints,

where elements of $\mathcal{C}$ are written as $V(a) \mapsto a$ (for keys) and $V(a) \subseteq V(b)$ (for inclusion constraints). Let $\Sigma = \{a_1, \ldots, a_n\}$.

Our algorithm to decide whether there exists a data tree $t \in \mathcal{L}(\mathcal{A})$ such that $t \models \mathcal{C}$ works as follows.

1. Construct the existential Presburger formula $\varphi_{\mathcal{A}}(x_1, \ldots, x_k)$ for the automaton $\mathcal{A}$ according to Lemma 4.2.
2. Let $\varphi_{\mathcal{C}}(x_1, \ldots, x_k)$ be the formula: $\exists v_1 \cdots \exists v_k$

$$\bigwedge_i v_i \leq x_i \quad \wedge \quad \bigwedge_i (v_i = 0 \leftrightarrow x_i = 0)$$

$$\wedge$$

$$\Big(\bigwedge_{V(a_i) \mapsto a_i \in \mathcal{C}} v_i = x_i \Big) \wedge \Big(\bigwedge_{V(a_i) \subseteq V(a_j) \in \mathcal{C}} v_i \leq v_j \Big).$$

3. Let $\varphi_{\mathcal{A},\mathcal{C}}(x_1, \ldots, x_k) := \varphi_{\mathcal{A}}(x_1, \ldots, x_k) \wedge \varphi_{\mathcal{C}}(x_1, \ldots, x_k)$.
   Test the satisfiability of $\varphi_{\mathcal{A},\mathcal{C}}(x_1, \ldots, x_k)$.

Note that here we do not use the variables z_S's.

We claim that for each data tree t, $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$ if and only if $\varphi_{\mathcal{A},\mathcal{C}}(\#a_1(t), \ldots, \#a_k(t))$ holds.

We start with the "only if" part. Let $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$. That $\varphi_{\mathcal{A}}(\#a_1(t), \ldots, \#a_k(t))$ follows from Lemma 4.2. To show that $\varphi_{\mathcal{C}}(\#a_1(t), \ldots, \#a_k(t))$ holds, we let the witnesses for each v_i as the cardinality $|V_t(a_i)|$, the number of data values found in the a_i-nodes in t. Then, it is straightforward to show that $\varphi_{\mathcal{C}}(\#a_1(t), \ldots, \#a_k(t))$ holds.

Now we show the "if" part. Suppose $\varphi_{\mathcal{A},\mathcal{C}}(n_1, \ldots, n_k)$ holds. By Lemma 4.2, there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that for each $a_i \in \Sigma$, $n_i = \#a_i(t)$. Let $(m_1, \ldots, m_k)$ be the witness for $(v_1, \ldots, v_k)$ that $\varphi_{\mathcal{C}}(x_1, \ldots, x_k)$ holds. We assign the values $1, \ldots, m_i$ as data values for the a_i-nodes in t such that $V_t(a_i) = \{1, \ldots, m_i\}$, for each $a_i \in \Sigma$. Such assignment is always possible since $m_i \leq \#a_i(t)$. That the keys and inclusion constraints in $\mathcal{C}$ are satisfied follows immediately from the constraints $v_i = x_i$ and $v_i \leq v_j$, respectively. □

This extends the results of [2, 13] which showed an NP bound for keys and a special form of inclusions (whose right-hand-sides are keys as well); but in addition our proof is much more streamlined compared to the proofs there.

Furthermore, it is easy to extend these results to more complex constraints studied in the XML context. For example, consider key constraints given by regular expressions over Σ. Such a constraint $V(e) \rightarrow e$, for a regular expression e, is satisfied in a tree t if nodes reachable from the root by following a path from e are uniquely determined by their data values. These constraints, common in XML schema specifications, are easily described by our formalism: one simply marks the nodes with states of an automaton for e, and uses the tree automaton $\mathcal{A}$ to ensure that the marking is correct.

6.2 Two-variable logic

As mentioned already, out main result does *not* follow from the decidability of the two-variable existential monadic second-order logic over data trees [6]. We now shall make precise the relationship between the two formalisms. When we talk about logics over data trees, we view them as structures

$$t = \langle D, E_{\downarrow}, E_{\rightarrow}, \{a(\cdot)\}_{a \in \Sigma}, \sim \rangle, \tag{2}$$

which extend unranked trees with the binary predicate $\sim$ interpreted as $v \sim v' \Leftrightarrow \mathrm{val}_t(v) = \mathrm{val}_t(v')$.

The sentences of the logic $\exists\mathrm{MSO}^2$ are of the form $\exists X_1 \ldots \exists X_m \, \psi$, where ψ is an FO formula over the vocabulary extended with the unary predicates $X_1, \ldots, X_m$ that uses only two variables, x and y. Every MSO sentence that does not mention data values is equivalent to an $\exists\mathrm{MSO}^2$ sentence. Other examples are keys ($\forall x \forall y \, (a(x) \wedge a(y) \wedge x \sim y \rightarrow x = y)$), inclusion constraints ($\forall x \exists y \, (a(x) \rightarrow b(y) \wedge x \sim y)$), and denial constraints ($\forall x \forall y \, (a(x) \wedge b(y) \rightarrow \neg(x \sim y))$).

It was shown in [6] that every $\exists\mathrm{MSO}^2$ formula over data trees is equivalent to a formula

$$\exists X_1 \ldots \exists X_k (\chi \wedge \bigwedge_i \varphi_i \wedge \bigwedge_j \psi_j)$$

where

1. χ describes a behavior of an automaton that can make "local" data comparisons (i.e., whether a data value in a node is equal/not equal the data value of its parent, left- or right-sibling);
2. each φ_i is of the form $\forall x \forall y (\alpha(x) \wedge \alpha(y) \wedge x \sim y \rightarrow x = y)$, where α is a conjunction of labeling predicates, X_k's, and their negations; and
3. each ψ_j is of the form $\forall x \exists y \, \alpha(x) \rightarrow (x \sim y \wedge \alpha'(y))$, with α, α' as in item 2.

If we extend the alphabet to $\Sigma \times 2^k$ so that each label also specifies the family of the X_i's the node belongs to, then formulae in items 2 and 3 can be encoded by constraints.

- Formulae in item 2 become conjunctions of keys and denial constraints over the extended alphabet. That is, it becomes a formula

  $$\forall x \forall y (\bigvee_{a \in \Sigma'} a(x) \wedge \bigvee_{a \in \Sigma'} a(y) \wedge x \sim y \rightarrow x = y),$$

  for some $\Sigma' \subseteq \Sigma \times 2^k$, which is equivalent to:
  - a is a key for each $a \in \Sigma'$, and
  - $V(a) \cap V(b) = \emptyset$, for every $a, b \in \Sigma'$ and $a \neq b$.
- Formulae in item 3 become

  $$\forall x \exists y (\bigvee_{a \in \Sigma'} a(x) \rightarrow x \sim y \wedge \bigvee_{a \in \Sigma''} a(y)),$$

for some $\Sigma', \Sigma'' \subseteq \Sigma \times 2^k$, which is equivalent to generalized inclusion constraints of the form

$$\bigcup_{a \in \Sigma'} V(a) \subseteq \bigcup_{b \in \Sigma''} V(b),$$

or, equivalently $\bigcup_{a \in \Sigma'} V(a) \cap \bigcap_{b \in \Sigma''} \overline{V(b)} = \emptyset$.

Hence, [6] and our results imply the following.

COROLLARY 6.2.
- (corollary of [6]) *Satisfiability of $\exists\mathrm{MSO}^2$ formulae over data trees is equivalent to satisfiability of keys, denial constraints, and generalized inclusions constraints with respect to an automaton with local data comparisons.*
- (corollary of Theorem 4.1) *Satisfiability of keys, denial constraints, and generalized inclusions constraints with respect to an automaton is solvable in* NP.

While our main result and the decidability of $\exists\mathrm{MSO}^2$ are incomparable, in essence we subsume $\exists\mathrm{MSO}^2$ *minus the local data comparison* constraints. Note that our proof is conceptually much simpler than the 30+ page proof of [6] that goes via more than a dozen reductions. Unlike [6], we fail to capture local data comparisons in automata; on the other hand, we add many properties (e.g., linear constraints) which are not even expressible in MSO.

7. CONCISE REPRESENTATIONS OF THE SATISFIABILITY PROBLEM

Recall that we measure the size of the linear data constraints $\mathbf{A}\bar{v} \geq \mathbf{b}$ as the sum of sizes of $\mathbf{A}$ and $\mathbf{b}$, with numbers represented in binary.

This could be a rather inefficient way of representing linear constraints. Since the number of variables z_S in the constraints is $2^{|\Sigma|-1} - 1$, we may achieve a more compact representation if only few of those variables are used in the constraints. Namely, we can safely disregard all the zero-columns in $\mathbf{A}$, and keep only the columns that correspond to variables actually used in constraints. This representation can be exponentially smaller than the full representation of the constraints (since Σ is a part of the input, we cannot achieve a smaller reduction even if there are no linear constraints).

We call this a *concise representation*, and consider the corresponding CONCISE-SAT$(\mathcal{A}, \mathcal{C})$: it is the same as the SAT$(\mathcal{A}, \mathcal{C})$ problem before, except we use a concise representation of linear constraints.

It is a very common phenomenon in complexity theory that going to concise representation increases the complexity by an exponent; in fact doing so is a common way of getting NEXPTIME-complete problems from NP-complete problems. Of course given a concise representation of constraints, we can always convert it into the usual representation in at most exponential time, and then apply Theorem 4.1. This immediately tells us that CONCISE-SAT is in NEXPTIME, and it is tempting to think that CONCISE-SAT is NEXPTIME-complete.

This, however, is not the case. Quite surprisingly, the concise representation does *not* increase the complexity of the problem. To show this, we need to design the decision procedure in a much more careful way.

THEOREM 7.1. *The problem* CONCISE-SAT$(\mathcal{A}, \mathcal{C})$ *is solvable in* NP.

We now indicate where the proof of Theorem 4.1 falls short when we have concise representations. First, the transformation from set to linear constraints is polynomial in the number of variables z_S, i.e., $O(2^{|\Sigma|})$. This did not cause problems before, but now we may not have all the variables z_S, so the input may be of the size $O(|\Sigma|^k)$ for a fixed k. Then the algorithm for eliminating set constraints becomes exponential. Second, the introduction of new variables v_a for $\sum_{a \in S \subseteq \Sigma} z_S$ used in the proof may likewise induce an exponential blow-up when considering concise representation.

The main aim is to show that *there exists a subset $\mathcal{Z} \subseteq 2^\Sigma$ of polynomial size such that there exists a tree $t \in \mathcal{L}(\mathcal{A})$ and $t \models \mathcal{C}$ iff there exists a tree $t' \in \mathcal{L}(\mathcal{A})$ and $t' \models \mathcal{C}$ and $[S]_{t'} = \emptyset$, for all $S \notin \mathcal{Z}$*. For this we introduce another extension of the ILP problem.

We give the sketch of the proof in the following subsection. The full proof will be available in the full version.

7.1 Sketch of Proof of Theorem 7.1

Let Σ be a finite alphabet and $\mathcal{C}$ is a collection of set and linear constraints. In the following we say that a term $\tau \in \mathcal{C}$ if and only if $\mathcal{C}$ contains a set constraint of the form $\tau = \emptyset$ or $\tau \neq \emptyset$. Similarly we say that a variable $z_S \in \mathcal{C}$ if and only if there is a linear data constraint in $\mathcal{C}$ that uses z_S. We denote by $\Psi_{\text{lin}}(\mathcal{C})$ the set of linear data constraints found in $\mathcal{C}$.

DEFINITION 7.2 ($\mathcal{C}$-FUNCTIONS). *Given an alphabet Σ and a collection $\mathcal{C}$ of data constraints, a $\mathcal{C}$-function is a function $\mathcal{F}$ from $\Sigma \cup \{\tau | \tau \in \mathcal{C}\} \cup \{ z_S | z_S \in \mathcal{C}\}$ to 2^Σ such that:*

- *for each $a \in \Sigma$, either $\mathcal{F}(a) = \emptyset$ or $a \in \mathcal{F}(a)$;*
- *for each $z_S \in \mathcal{C}$, either $\mathcal{F}(z_S) = \emptyset$ or $\mathcal{F}(z_S) = S$;*
- *for each constraint $\tau \neq \emptyset \in \mathcal{C}$, we have $\mathcal{F}(\tau) \in \mathbb{S}(\tau)$;*
- *for each constraint $\tau = \emptyset \in \mathcal{C}$, we have $\mathcal{F}(\tau) = \emptyset$ and $Im(\mathcal{F}) \cap \mathbb{S}(\tau) = \emptyset$;*

where $Im(\mathcal{F})$ denotes the image of $\mathcal{F}$, and $\mathbb{S}(\tau)$ was defined in the proof of Proposition 8.4.

The intuition of $\mathcal{F}$ is such that $Im(\mathcal{F})$ is the desired set $\mathcal{Z}$. Given a collection $\mathcal{C}$ of data constraints and a $\mathcal{C}$-function $\mathcal{F}$, we call $\Psi(\mathcal{C}, \mathcal{F})$ the system obtained from $\mathcal{C}$ by adding the following constraints to $\Psi_{\text{lin}}(\mathcal{C})$:

$z_S \geq 1$		for each $S \in Im(\mathcal{F}) - \emptyset$
$x_a = 0$		for each $a \in \Sigma$ such that $\mathcal{F}(a) = \emptyset$
$z_S = 0$		for each $z_S \in \mathcal{C}$ such that $\mathcal{F}(z_S) = \emptyset$
$\sum_{a \in S \in Im(\mathcal{F}) - \emptyset}$	$z_S \leq x_a$	for each $a \in \Sigma$;

Notice that the size of $\Psi(\mathcal{C}, \mathcal{F})$ is polynomial in the size of both $\mathcal{C}$ and the alphabet Σ.

In the rest of the proof, all instances of ILP we refer to are instances over the variables x_a, z_S, v_a.

DEFINITION 7.3 (ILP UNDER $\mathcal{C}$-CONDITION). *An instance of ILP problem under $\mathcal{C}$-condition is given by an instance Ψ of ILP together with a collection $\mathcal{C}$ of data constraints. We say that it has a non-negative solution if there exists a $\mathcal{C}$-function $\mathcal{F}$ such that the instance of ILP given by Ψ and $\Psi(\mathcal{F}, \mathcal{C})$ has a non-negative solution.*

We shall now state the two main lemmas from which Theorem 7.1 follows immediately. The proofs will be available in the full version.

LEMMA 7.4. *Checking whether an instance of ILP with $\mathcal{C}$-condition has a non-negative solution can be done in* NP.

LEMMA 7.5. *Given an automaton $\mathcal{A}$ and a set $\mathcal{C}$ of data constraints in concise representation, one can construct, in polynomial time, an instance of ILP with $\mathcal{C}$-condition so that there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $t \models \mathcal{C}$ iff the instance of ILP with $\mathcal{C}$-conditions has a non-negative solution.*

8. CONVERTING AUTOMATA TO PRESBURGER FORMULA

To make our proof completely algorithmic, in this section we spell out the translation from automata to a Presburger formula defining Parikh images of trees, used as a black box (Lemma 4.2) in the proof of Theorem 4.1. Moreover, we also present an algorithm, that given a solution to the Presburger formula, constructs a tree accepted by the original automaton.

We recall that the Parikh image of a tree t over $\Sigma = \{a_l, \ldots, a_n\}$ is an n-tuple $\mathsf{Parikh}(t) = (\#a_1(t), \ldots, \#a_n(t))$, and the Parikh image of a tree language L is $\mathsf{Parikh}(L) = \{\mathsf{Parikh}(t) \mid t \in L\} \subseteq \mathbb{N}^n$.

PROPOSITION 8.1. *There is a quadratic time algorithm that, given an unranked tree automaton $\mathcal{A}$ over $\Sigma = \{a_l, \ldots, a_n\}$, returns a formula*

$$\varphi_{\mathcal{A}}(x_1, \ldots, x_n) \;=\; \exists \bar{y}\; \alpha(\bar{x}, \bar{y})$$

of at most quadratic size such that

- *if $t \in \mathcal{L}(\mathcal{A})$, then $\varphi_{\mathcal{A}}(\#a_1(t), \ldots, \#a_n(t))$ holds; and conversely,*
- *if $\varphi_{\mathcal{A}}(k_1, \ldots, k_n)$ holds, then there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that $\#a_1(t) = k_1, \ldots, \#a_n(t) = k_n$ and α is a conjunction of formulae of the form:*
  - $\mathbf{A}(\bar{x}, \bar{y}) \geq \mathbf{b}$, *where all the entries of* $\mathbf{A}$ *and* $\mathbf{b}$ *are either* 0 *or* 1 *or* -1*;*
  - *formulae $(w = 0 \vee u \geq 1)$ where w, u are variables among $\bar{x}, \bar{y}$; and*
  - *disjunctions $\bigvee_i (w_i \geq 1 \wedge u_i = 1)$, where w_i's and u_i's are variables among $\bar{x}, \bar{y}$.*

Moreover, from every solution $(k_1, \ldots, k_n)$ and witness tuple $\bar{m}$ such that $\alpha(k_1, \ldots, k_n, \bar{m})$ holds, we can construct effectively a tree $t \in \mathcal{L}(\mathcal{A})$ such that $\mathsf{Parikh}(t) = (k_1, \ldots, k_n)$.

8.1 Proof of Proposition 8.1

The general outline is as follows: we first replace an automaton by an extended DTD (Proposition 8.2), and then by a DTD of a special form, which we call *simple* DTD (Proposition 8.3). We then show the construction of the Presburger formula for such simple DTDs (Proposition 8.4). The first two reductions are standard. The crucial one is the last one.

Recall that an extended document type definition (EDTD) is a context-free grammar in which the right-hand sides of productions can be regular expressions. Formally, an extended DTD over the alphabet $(\Gamma \cup \Lambda)$ of nonterminals Γ, with a distinguished symbol r for the root, and terminals Λ is $\mathfrak{G} = (\Gamma, \Lambda, \Delta)$, where Δ assigns to each symbol $a \in \Gamma$ a regular expression over $(\Gamma \cup \Lambda) - \{r\}$. The set of trees of $\mathfrak{G}$ is denoted by $\mathcal{T}(\mathfrak{G})$. That is, an unranked tree t is in $\mathcal{T}(\mathfrak{G})$ if its root is labeled r, for each node v labeled $a \in \Gamma$ with children $v \cdot 0, \ldots, v \cdot (n-1)$, the word of their labels, i.e., $\ell ab_t(v \cdot 0) \cdots \ell ab_t(v \cdot (n-1))$, is in the language of $\Delta(a)$, and each node labeled with $b \in \Lambda$ is a leaf.

The first reduction is stated as a proposition below. The proof will be available in the full version.

PROPOSITION 8.2. *Given an automaton $\mathcal{A}$ with the set Q of states over alphabet Σ, one can construct, in quadratic time, an extended DTD $\mathfrak{G} = (\Gamma, \Sigma - \{r\}, \Delta)$ with $\Gamma = Q \times \Sigma \cup \{r\}$ such that the following holds.*

1. *For all tree $t \in \mathcal{L}(\mathcal{A})$, there exists a tree $t' \in \mathcal{T}(\mathfrak{G})$ such that for all $a \in \Sigma$, $\#a(t) = \#a(t')$.*
2. *Vice versa, for all tree $t' \in \mathcal{T}(\mathfrak{G})$, there exists a tree $t \in \mathcal{L}(\mathcal{A})$ such that for all $a \in \Sigma$, $\#a(t) = \#a(t')$.*

Moreover, every tree $t' \in \mathcal{T}(\mathfrak{G})$ can be converted effectively into a tree $t \in \mathcal{L}(\mathcal{A})$.

Next, we define *simple DTDs* as $\mathcal{G} = (\Gamma, \Lambda, \Delta)$ with a designated terminal symbol $\lambda \in \Lambda$. In them, $\Delta(a)$ is one of the following: b, or bc, or $b|c$, or λ, where $b, c \in (\Gamma \cup \Lambda) - \{r\}$. We denote the set of parse trees of $\mathcal{G}$ by $\mathcal{T}(\mathcal{G})$. Note that trees in $\mathcal{T}(\mathcal{G})$ can have only unary or binary branching. We make the standard assumption that all symbols in Γ are reachable from the root symbol r. If a CFG has unreachable symbols, they can be eliminated without affecting the set $\mathcal{T}(\mathcal{G})$.

The second reduction is stated as proposition below. The proof will be available in the full version.

PROPOSITION 8.3. *Given an extended DTD $\mathfrak{G} = (\Gamma, \Lambda, \Delta)$, one can construct, in linear time, a simple DTD $\mathcal{G} = (\Gamma', \Lambda \cup \{\lambda\}, \Delta')$ such that the following holds.*

1. *For all tree $t \in \mathcal{T}(\mathfrak{G})$, there exists a tree $t' \in \mathcal{T}(\mathcal{G})$ such that for all $a \in \Lambda$, $\#a(t) = \#a(t')$.*
2. *Vice versa, for all tree $t' \in \mathcal{T}(\mathcal{G})$, there exists a tree $t \in \mathcal{T}(\mathfrak{G})$ such that for all $a \in \Lambda$, $\#a(t) = \#a(t')$.*

Moreover, every tree $t' \in \mathcal{T}(\mathcal{G})$ can be converted effectively into the tree $t \in \mathcal{T}(\mathfrak{G})$.

The last reduction is stated as proposition below.

PROPOSITION 8.4. *Given a simple DTD $\mathcal{G} = (\Gamma, \Lambda \cup \{\lambda\}, \Delta)$, where $\Lambda = \{a_1, \ldots, a_n\}$, one can construct, in linear time, an existential Presburger formula $\varphi_{\mathcal{G}}(x_1, \ldots, x_n) := \exists \bar{y} \psi(\bar{x}, \bar{y})$ such that for every tree t, $t \in \mathcal{T}(\mathcal{G})$ iff $\varphi_{\mathcal{G}}(\#a_1(t), \ldots, \#a_n(t))$ holds.*
Moreover, from every solution $(k_1, \ldots, k_n)$ and $\bar{m}$ such that $\psi(k_1, \ldots, k_n, \bar{m})$ holds, we can construct effectively a tree $t \in \mathcal{T}(\mathcal{G})$ such that $\mathsf{Parikh}(t) = (k_1, \ldots, k_n)$.

We devote the rest of this subsection to the proof of Proposition 8.4. We need a new notation here. For a tree t over the alphabet $\Gamma \cup \Lambda \cup \{\lambda\}$, we define a *directed* graph $G_t = (V_t, E_t)$, where the set of vertices is $V_t = \Gamma \cup \Lambda \cup \{\lambda\}$; and for every $a, b \in \Gamma \cup \Lambda \cup \{\lambda\}$, there is an edge $(a, b) \in E_t$ if there exists a node in t labeled with b and whose parent is labeled with a. If a symbol a does not appear in the tree t, then it is an isolated vertex in G_t.

The main idea is to prove that a tree $t \in \mathcal{T}(\mathcal{G})$ iff the following quantities:

1. $n_a = \#a(t)$, for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$;
2. $n_{a\downarrow b}$ is the number of b-nodes whose parents in t is labeled with a;
3. δ_a is the length of *some* path from the root r to the symbol a in the graph G_t,

satisfy the following relations:

- $n_a = \sum_{b \in \Gamma \cup \Lambda} n_{b\downarrow a}$, for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$.
- – $n_a = n_{a\downarrow b} + n_{a\downarrow c}$, if $\Delta(a) = b|c$.
  – $n_a = n_{a\downarrow b} = n_{a\downarrow c}$, if $\Delta(a) = bc$,
  – $n_a = n_{a\downarrow b}$, if $\Delta(a) = b$,
- $\delta_r = 0$;
- for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$ and $a \neq r$,
$$\delta_a = -1 \leftrightarrow n_a = 0$$
and
$$\bigvee_{n_{b\downarrow a} \neq 0 \text{ and } \delta_b \neq -1} \delta_a = \delta_b + 1$$

Note that by default, we set $\delta_a = -1$, if there is no path from the root to the symbol a in the graph G_t, which means that the symbol a does not appear in t.

Then, the construction of the desired formula $\varphi_{\mathcal{G}}$ is straightforward. It uses the variables x_a's, y_a's and $x_{a\downarrow b}$'s, for all $a \in \Gamma \cup \Lambda$ and b appears in $\Delta(a)$. The intended meaning of each variable is as follows: x_a is for n_a; $x_{a\downarrow b}$ is for $n_{a\downarrow b}$; y_a is for δ_a.

The formula $\varphi_{\mathcal{G}}$ is the conjunction of the following:

- $x_r = 1$;
- $x_a = \sum_{b \in \Gamma \cup \Lambda} x_{b\downarrow a}$ for each $a \in \Gamma \cup \Lambda$;
- $x_a = x_{a\downarrow b} = x_{a\downarrow c}$ for each $\Delta(a) = bc$;
- $x_a = x_{a\downarrow b} + x_{a\downarrow c}$ for each $\Delta(a) = b|c$;
- $x_a = x_{a\downarrow b}$ for each $\Delta(a) = b$;
- $y_r = 0$;
- for each $a \in \Gamma \cup \Lambda \cup \{\lambda\}$,
$$y_a = -1 \leftrightarrow x_a = 0$$
$$\vee$$
$$\bigvee_{a \text{ appears in } \Delta(b)} y_a = y_b + 1 \wedge x_{b\downarrow a} \neq 0 \wedge y_b \neq -1.$$

The total number of variables x_a's and $x_{a\downarrow b}$'s and y_a's is linear in the size of Δ. We do not need the variables $x_{a\downarrow b}$'s, if b does not appear in $\Delta(a)$.

By existentially quantifying all the variables $x_{a\downarrow b}$'s and y_a's, we can then view $\varphi_{\mathcal{G}}$ as an existential Presburger formula with x_a's as the free variables.

Proposition 8.4 follows immediately from the lemma below.

LEMMA 8.5. *Let $\mathcal{G} = (\Gamma, \Lambda, \Delta)$ be a simple CFG. The formula $\varphi_{\mathcal{G}}(\bar{n})$ holds – where $(\bar{n}) = (n_a)_{a \in \Sigma}$ and the witnesses for $x_{a\downarrow b}$'s and y_a's are: $x_{a\downarrow b} = n_{a\downarrow b} \in \mathbb{N}$, and $y_a = d_a \in \mathbb{N}$, for $a, b \in \Gamma \cup \Lambda$ – if and only if there exists a tree $t \in \mathcal{T}(\mathcal{G})$ such that*

(1) $n_a = \#a(t)$ for each $a \in \Gamma \cup \Lambda$,
(2) $n_{a\downarrow b}$ is the number of b-nodes whose parents are a-nodes, and
(3) d_a is the length of some path from the root r to the symbol a in the graph G_t.

PROOF. From the definition of $\Psi(\mathcal{G})$, the "if" part is immediate. We prove the other implication. Let $\bar{n} = (n_a)_{a \in \Sigma}$ such that $\varphi_{\mathcal{G}}(\bar{n})$ holds. Let $n_{a\downarrow b}$ be the witness for $x_{a\downarrow b}$ for $a, b \in \Gamma \cup \Lambda$, and d_a for y_a, for $a \in \Gamma \cup \Lambda$. Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be a directed graph where the set $\tilde{V}$ of nodes is $\Gamma \cup \Lambda$ and the set $\tilde{E}$ of edges is defined as: $(a, b) \in \tilde{E}$ iff $n_{a\downarrow b} \neq 0$.

We shall construct a tree $t \in \mathcal{T}(\mathcal{G})$ that satisfies (1) and (2) and that $G_t = \tilde{G}$. First, we construct a *directed* graph $G = (V, E)$ with the following properties.

(i) For each $a \in \Gamma \cup \Lambda$, there are exactly n_a nodes labeled a.

(ii) For each $a, b \in \Gamma \cup \Lambda$, there are exactly $n_{a\downarrow b}$ edges going from an a-node to a b-node.

(iii) There is exactly one node labeled r and it has no incoming edges (the root node).

(iv) All nodes, except the root node, have exactly one incoming edge.

(v) For all nodes, outgoing edges conform to Δ. That is, for each $a \in \Gamma$, the outgoing edges from a-nodes are as follows: if $\Delta(a) = b \cdot c$, there are exactly two outgoing edges: one to a b-node and one to a c-node; if $\Delta(a) = b|c$, there is exactly one outgoing edge going to a node labeled by b or c; and if $\Delta(a) = b$, there is exactly one outgoing edge that goes to a b-node.

Procedure 1 shows the construction of the graph G. Since $n_r = 1$, there is only one root node in G. The steps 7, 8, 12, 13 and 17 of the procedure are possible due to the equality $x_b = \sum_{d \in \Gamma \cup \Lambda} x_{d\downarrow b}$ in $\Psi(\mathcal{G})$. Properties (i)-(v) follow directly from the construction and the constraints given in $\Psi(\mathcal{G})$.

Procedure 1 Construct Graph $G = (V, E)$

1: The set V consists of $\sum_{a \in \Gamma \cup \Lambda} n_a$ nodes.
   For each $a \in \Gamma \cup \Lambda$, we label n_a nodes with a.
2: $E := \emptyset$.
3: **for all** $a \in \Gamma$ **do**
4: Let $w_1, \ldots, w_{n_a}$ be the a-nodes.
5: **if** $\Delta(a) = b \cdot c$ **then**
6: Let $n = n_a = n_{a\downarrow b} = n_{a\downarrow c}$.
7: Pick a sequence $u_1, \ldots, u_n$ of n distinct b-nodes with no incoming edges in E.
8: Pick a sequence $v_1, \ldots, v_n$ of n distinct c-nodes with no incoming edges in E.
9: $E := E \cup \{(w_i, u_i), (w_i, v_i)\}_{i=1,\ldots,n}$.
10: **end if**
11: **if** $\Delta(a) = b \cup c$ **then**
12: Pick a sequence $u_1, \ldots, u_{n_{a\downarrow b}}$ of $n_{a\downarrow b}$ distinct b-nodes with no incoming edges in E.
13: Pick a sequence $v_1, \ldots, v_{n_{a\downarrow c}}$ of $n_{a\downarrow c}$ distinct c-nodes with no incoming edges in E.
14: $E := E \cup \{(w_i, u_i)\}_{i=1,\ldots,n_{a\downarrow b}} \cup \{(w_{n_{a\downarrow b}+j}, v_j)\}_{j=1,\ldots,n_{a\downarrow c}}$.
15: **end if**
16: **if** $\Delta(a) = b$ **then**
17: Pick a sequence $u_1, \ldots, u_{n_{a\downarrow b}}$ of $n_{a\downarrow b}$ distinct b-nodes with no incoming edges in E.
18: $E := E \cup \{(w_i, u_i)\}_{i=1,\ldots,n_{a\downarrow b}}$.
19: **end if**
20: **end for**

If G were a tree, we would be done: membership in $\mathcal{T}(\mathcal{G})$ would follow from (v), property (1) from (i), and property (2) from (ii) and (v). Therefore, to finish the proof of Lemma 8.5, we show Claim 8.6 and Claim 8.8 below.

CLAIM 8.6. *A connected directed graph $G = (V, E)$ that satisfies (i)-(v) is a tree.*

PROOF. From Properties (iii) and (iv), we can see that the graph G satisfies the equation $|E| = |V| - 1$. If we forget about orientation, this equation implies that a connected graph is a tree [35]. The root (the r-labeled node) gives the tree a unique orientation; we must show that it is the same one as the one in G. For this, consider any path from the root to a leaf in the tree, and suppose one edge has an orientation different from G. Let (u, v) be the first such edge; that is, in G we have an edge (v, u). This cannot be the first edge of the path, as the root has no incoming edge in G. Hence u has a parent u' in the oriented tree, and the edge (u', u) has the same orientation in both the oriented tree and in G. But this tells us that u has two incoming edges, which contradicts (iv). □ □

We shall use Claim 8.7 to prove Claim 8.8.

CLAIM 8.7. *In the directed graph $\tilde{G}$, a node a is connected from the root symbol r iff $d_a \neq -1$, or equivalently, $n_a \neq 0$. Moreover, d_a is the length of some path from the root symbol r to a, if $d_a \neq -1$.*

PROOF. The proof is by straightforward induction on the value d_a. The base case, $d_a = 0$, is trivial as it means $a = r$. The induction hypothesis is that for each node a with $d_a = k \neq -1$ is connected from the root symbol r by a path of length k.

Suppose b is a node such that $d_b = k + 1$. By the construction of $\varphi_{\mathcal{G}}$, there exists a node a such that $n_{a\downarrow b} \neq 0$ and $d_a = k$. By the induction hypothesis, a is connected from the root symbol r with a path of length k, and by the construction of $\tilde{G}$, there exists an edge from a to b. Thus, our claim holds. □

CLAIM 8.8. *From a directed graph $G = (V, E)$ that satisfies (i)-(v), one can compute in polynomial time a connected directed graph $G' = (V, E')$ that also satisfies (i)-(v).*

PROOF. The idea is to change a few edges in G in order to connect all components to the connected component that contains the r-node. We first observe the following. Suppose G consists of several connected components: $G_0, G_1, \ldots, G_l$, where G_0 is the component that contains the root node. Then, there exist a node u in G_0 and a node v in one of $G_1, \ldots, G_l$ such that u and v are labeled by the same symbol from Σ.

By Claim 8.7, if $d_a \neq -1$ (thus, $n_a \neq 0$), the symbol a is connected to the root symbol r in $\tilde{G}$, and that d_a is the length of some path from r to a in $\tilde{G}$. So, for every symbol a that appears in G, there exists a sequence of symbols $b_0, b_1, \ldots, b_j$, respectively, where

- $b_0 = r$,
- $b_l = a$, and
- for each $i = 0, \ldots, j-1$, $n_{b_i\downarrow b_{i+1}} \neq 0$.

If the symbol a does not appear in G_0, then there are a node u in G_0 and a node v in one of $G_1, \ldots, G_l$ such that both u and v are labeled with the same symbol $b_i \in \{b_1, \ldots, b_l\}$.

Let G_1 be the component that contains that node v. By (v), the node v has as many children as u (and it has at least one child as it is not labeled by λ).

If u and v have one child each, then let w_1 and w_2 be their respective children. We can then connect G_0 and G_1 by replacing the edges (u, w_1) and (v, w_2) with (u, w_2) and (v, w_1). If u and v have two children each, then let w_1, w_1' and w_2, w_2' be their respective children. We can then connect G_0 and G_1 by replacing the edges $(u, w_1), (u, w_1')$ and $(v, w_2), (v, w_2')$ with $(u, w_2), (u, w_2')$ and $(v, w_1), (v, w_1')$.

It is straightforward to see that after such edge replacement the graph still satisfies properties (i)-(v), and each edge replacement reduces the number of connected components, so eventually this algorithm produces a tree t that satisfies (i)-(v). Moreover, the numbers $n_{a\downarrow b}$ do not change during the process, thus, $G_t = \tilde{G}$. □

This completes the proof of Lemma 8.5. □

9. CONCLUSIONS

We have studied the consistency problem of set and linear constraints with respect to regular tree languages given by an automata. We prove that this problem is solvable in NP (the hardness result has been known even for simple keys and inclusion constraints [13]).

At least as important as the result itself, we provide an original and modular proof using simple proof techniques as in particular linear integer programming. This unable us to extend the result to more complicated path constraints. Surprisingly, we can use the same techniques to show that the complexity of the problem remains NP even when considering concise representation of the constraints.

In terms of expressivity, our formalism subsumes many interesting formalism such as keys, inclusions and denial constraints. Our formalism is also closely related to the extension of $\exists\mathrm{MSO}^2$ presented in [6], as it subsumes $\exists\mathrm{MSO}^2$ *minus the local data comparison* constraints. In addition, it is also able to express many non-MSO properties.

Acknowledgment. We thank the anonymous referees for their comments. This work was supported by the FET-Open project FoX (Foundations of XML), grant agreement FP7-ICT-233599, and by EPSRC grant G049165. This work was done when the first author was at the University of Edinburgh.

10. REFERENCES

[1] N. Alon, T. Milo, F. Neven, D. Suciu, V. Vianu. XML with data values: typechecking revisited. *J. Comput. Syst. Sci.* 66(4): 688-727 (2003).
[2] M. Arenas, W. Fan, L. Libkin. On the complexity of verifying consistency of XML specifications. *SIAM J. Comput.* 38(3): 841-880 (2008).
[3] M. Arenas, L. Libkin. XML data exchange: consistency and query answering. *J. ACM* 55(2): (2008).
[4] H. Björklund, M. Bojanczyk. Bounded depth data trees. In *ICALP'07*, pages 862–874.
[5] H. Björklund, W. Martens, T. Schwentick. Optimizing conjunctive queries over trees using schema information. *MFCS'08*, pages 132–143.
[6] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM* 56(3): (2009).
[7] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on words with data. In *LICS'06*, pages 7-16.
[8] P. Bouyer, A. Petit, D. Thérien. An algebraic characterization of data and timed languages. *CONCUR'01*, pages 248–261.
[9] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Regular XPath: constraints, query containment and view-based answering for XML documents. In *LID'08*, 2008.
[10] H. Comon, et al. *Tree Automata: Techniques and Applications*. October 2007.
[11] S. Dal-Zilio, D. Lugiez, C. Meyssonnier. A logic you can count on. *POPL 2004*, 135–146.
[12] S. Demri, R. Lazic. LTL with the freeze quantifier and register automata. *ACM TOCL* 10(3): (2009).
[13] W. Fan, L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM* 49(3): 368–406 (2002).
[14] D. Figueira. Satisfiability of downward XPath with data equality tests. *PODS'09*, 197-206.
[15] R. Givan, D. A. McAllester, C. Witty, and D. Kozen. Tarskian set constraints. *Inform. and Comput.*, 174 (2002), pp. 105–131.
[16] M. Jurdzinski, R. Lazic. Alternation-free modal mu-calculus for data trees. In *LICS'07*, pages 131–140.
[17] M. Kaminski, T. Tan. Tree automata over infinite alphabets. In *Pillars of Computer Science*, 2008, pages 386–423.
[18] E. Kopczynski, A. Widjaja To. Parikh Images of Grammars: Complexity and Applications. In *LICS 2010*.
[19] L. Libkin, C. Sirangelo. Reasoning about XML with temporal logics and automata. *J. Applied Logic*, 8:2, 210–232 (2010).
[20] S. Malik and L. Zhang. Boolean satisfiability: from theoretical hardness to practical success. *CACM* **52**(8), 76–82, 2009.
[21] W. Martens, F. Neven, T. Schwentick. Simple off the shelf abstractions for XML schema. *SIGMOD Record* 36(3): 15-22 (2007).
[22] T. Milo, D. Suciu, V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.* 66(1): 66-97 (2003).
[23] F. Neven. Automata, logic, and XML. In *CSL 2002*, pages 2–26.
[24] F. Neven, Th. Schwentick. Query automata over finite trees. *Theor. Comput. Sci.* 275(1-2):633-674 (2002).
[25] F. Neven, Th. Schwentick, V. Vianu. Towards regular languages over infinite alphabets. *MFCS 2001*, pages 560–572.
[26] L. Pacholski, A. Podelski. Set constraints: a pearl in research on constraints. In *CP'97*, pages 549–562.
[27] C. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28 (1981), 765–768.
[28] A. Robinson, A. Voronkov, eds. *Handbook of Automated Reasoning*. The MIT Press, 2001.
[29] Th. Schwentick. XPath query containment. *SIGMOD Record* 33(1): 101-109 (2004).
[30] H. Seidl, Th. Schwentick, A. Muscholl. Numerical document queries. *PODS'03*, 155–166.
[31] H. Seidl, Th. Schwentick, A. Muscholl, P. Habermehl. Counting in trees for free. In *ICALP 2004*, pages 1136–1149.
[32] J.W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *JCSS* 1 (1967), 317–322.
[33] K. N. Verma, H. Seidl, T. Schwentick On the Complexity of Equational Horn Clauses. In *CADE 2005*, pages 337–352.
[34] V. Vianu. A web Odyssey: from Codd to XML. In *PODS'01*, pages 1–15.
[35] D. West. *Introduction to Graph Theory*. Prentice Hall, 2001.